Examen PP varianta B — NOT EXAM MODE

31.05.2019

ATENȚIE: Aveți 2 ore · 1-9: 10p; 10: 30p · 100p pentru nota maximă · **Justificați** răspunsurile!

---

1. Reduceți expresia lambda $E = (y\ \ (\lambda x.\lambda x.x\ \ (\lambda y.y\ \ y)))$
   Soluție:
   $\rightarrow (y\ \ (\lambda x.\lambda x.x\ \ y)) \rightarrow (y\ \ \lambda x.x)$
   sau
   $\rightarrow (y\ \ \lambda x.x)$

2. Se dă următorul cod Racket:
```
(define computation (λ () (equal? 5 5)))
(define (f x) (and (> x 5) (computation)))
(filter f '(1 3 5 7 9))
```
   (a) De câte ori se apelează funcția `equal?` ?
   (b) Rescrieți codul pentru `computation` și pentru `f` folosind promisiuni (pentru întârzierea lui `computation`) și răspundeți din nou la întrebarea (a).
   Soluție:
   (a) de 2 ori (pentru fiecare element mai mare decât 5)
   (b)
```
(define computation (delay (equal?  5 5)))
(define (f x) (and (> x 5) (force computation)))
(filter f '(1 3 5 7 9))
```
   acum se apelează o singură dată, la prima evaluare a lui `computation`.

3. Dată fiind o listă de liste de numere `LL`, scrieți în Racket codul care produce sublista lui `LL` în care pentru toate elementele `L` suma elementelor este cel puțin egală cu produsul lor. E.g. pentru `L = ((1 2 3) (1 2) (4 5) (.5 .5))` rezultatul este `((1 2 3) (1 2) (0.5 0.5))`. Nu folosiți recursivitate explicită.
   Soluție:
```
(filter (lambda (L) (>= (apply + L) (apply * L))) '((1 2 3) (1 2) (4 5) (.5 .5)))
```

4. Sintetizați tipul următoarei funcții în Haskell: `f = map (++)`
   Soluție:
```
map ::  (a -> b) -> [a] -> [b]
(++) ::  [c] -> ([c] -> [c])
a = [c]
b = [c] -> [c]
f ::  [[c]] -> [[c] -> [c]]
```

5. (a) Câte aplicații ale funcției de incrementare sunt calculate pentru evaluarea expresiei Racket `(length (map add1 '(1 2 3 4 5 6 7 8 9 10)))` ?
   (b) Dar pentru expresia Haskell `length $ map (+ 1) [1 .. 10]` ?
   Soluție:
   (a) Toate elementele listei sunt evaluate, deci 10.
   (b) Elementele listei nu sunt evaluate, deci 0.

6. Supraîncărcați în Haskell operatorii `(+)` și `(*)` pentru valori booleene, pentru a surprinde operațiile de *sau*, respectiv *și* logic.
   Soluție:
```
instance Num Bool where
  (+) = (||)
  (*) = (&&)
```

7. Transformați propoziția „Nu mor caii când vor câinii.” în logică cu predicate de ordinul întâi, folosind predicatele `caii_mor(când)` și `câinii_vor(când)`.
   Soluție:
   $\exists t.cainii\_vor(t) \wedge \neg caii\_mor(t)$ – există și momente când câinii vor dar caii nu mor
   sau
   $\neg(\forall t.cainii\_vor(t) \Rightarrow caii\_mor(t))$ – nu este adevărat că oricând câinii vor automat caii mor

8. Se dă programul Prolog:
   ```
   p(_, [], []).
   p(A, [A|B], B) :- !.
   p(A, [B|C], [B|D]) :- p(A, C, D).
   ```
   Ce relație există între cele 3 valori `X`, `Y`, `Z`, dacă `p(X, Y, Z)` este adevărat?
   Soluție:
   Este predicatul `select`, iar dacă primul argument este nelegat face `select` la primul element. Predicatul `select(X, Y, Z)` este adevărat dacă `X` este un element din lista `Y`, iar `Z` este exact lista `Y`, în afară de elementul `X`.

9. Se dau următoarele relații genealogice prin predicatul `c(Parinte, Copil)`. Implementați predicatul `veri(X, V)`, care leagă `V` la lista de veri ai lui `X` (dacă există). De exemplu, pentru definițiile de mai jos, interogarea `veri(faramir, V)` leagă `V` la `[jenny, karl, ninel, octav]`.
   ```
   c(alex, celia).  c(alex, delia).  c(alex, marcel).
   c(barbra, celia).  c(barbra, delia).  c(barbra, marcel).
   c(delia, faramir).
   c(ion, jenny).  c(ion, karl).  c(celia, jenny).  c(celia, karl).
   c(marcel, ninel).  c(marcel, octav).
   ```
   Soluție:
   ```
   veri(X, L) :- setof(V, P^B^U^(c(P, X), c(B, P), c(B, U), U\=P, c(U, V)), L). sau
   veri(X, L) :- findall(V, (c(P, X), c(B, P), c(B, U), U\=P, c(U, V)), L1), sort(L1, L).
   ```

10. PROBLEMA (Poate fi implementată în orice limbaj studiat la PP.) Se urmărește implementarea unui *hash set*, care reprezintă o mulțime grupând valorile în bucket-uri, fiecare bucket fiind unic determinat de hash-ul valorilor din bucket (toate valorile din bucket au același hash). Hashul unei valori va fi dat de funcția `hash`, respectiv predicatul `hash(+V, -Hash)`.
    (a) Descrieți reprezentarea *hash set*-ului. Pentru Haskell, dați definiția tipului de date polimorfic. Definiți funcția/predicatul `values'`, care extrage lista tuturor valorilor asociate cu un *hash*.
    (b) Definiți funcția/predicatul `insert'`, pentru adăugarea unei valori.
    (c) Definiți funcția/predicatul `map'`, care aplică o funcție/predicat pe fiecare valoare din *hash-set*. NOTĂ: în Prolog, `map'` va aplica întotdeauna un același predicat `p(+VIn, -VOut)`.
    Soluție:
    Racket:
    ```
    (define (hash int) (remainder int 2))
    (define hashExample '((1 3 5 7) (0 0 2 8)))

    (define (lookup hash m) (cdr (assoc hash m)))
    (lookup 1 hashExample)

    (define (insert v m)
      (let ((k (hash v)))
        (let-values (((bef aft) (splitf-at m (λ(kv) (not (equal?  (car kv) k))))))
          (if (null?  aft)
            (cons (list k v) m)
            (append bef (list (cons k (cons v (cdar aft)))) (cdr aft))
            ))))
    (insert 9 hashExample)
    (insert 9 (cdr hashExample))

    (define (mmap f m)
    ```

```
  (map
    (λ(kv) (cons (car kv) (map f (cdr kv))))
   m))
 (mmap add1 hashExample)
```

Haskell:

```
class Hashable a where hash ::  a -> Int -- nu a fost cerut în rezolvare
instance Hashable Int where hash x = mod x 2 -- nu a fost cerut în rezolvare

data HashSet a = HS [(Int, [a])] deriving (Eq, Show)

--ins ::  Hashable a => a -> HashSet a -> HashSet a
 --// tipul corect, dar mai greu de testat
ins ::  Int -> HashSet Int -> HashSet Int
ins a (HS lst) = HS $ case back of
  [] -> (k, [a]) :  front
  (_, as) :  back -> (k, a :  as) :  front ++ back
  where
    k = hash a
    (front, back) = break ((== k) .  fst) lst

map ::  (Hashable a, Hashable b) => (a -> b) -> HashSet a -> HashSet b
map f (HS lst) = HS $ map (\(k, as) -> (k, map f as)) lst

test2 = ins 5 $ ins 2 $ ins 4 $ ins 8 $ HS []
test3 = map (+1) $ test2
```

Prolog:

```
hash(V, Hash) :- Hash is mod(V, 2).

lookup(Hash, HS, Values) :- member((Hash, Values), HS).

insert(V, HS, Out) :-
  hash(V, K),
  select((K, L), HS, HS1), !,
  Out = [(K, [V|L]) | HS1].
insert(V, HS, [(K, [V]) | HS]) :- hash(V, K).

% insert(5, [], H1), insert(7, H1, H2), insert(2, H2, H3).

f(V, V1) :- V1 is V + 1.

map(HS, Out) :-
  findall((K, L1),
    ( member((K, L), HS),
      findall(E1, (member(E, L), f(E, E1)), L1)),
    Out).
```